



# Corso avanzato

Fabrizio Rizzi e  SORINT<sub>lab</sub>

# About me

Ciao, mi chiamo Fabrizio e sono uno sviluppatore web,  
specializzato in tecnologie front-end.

Sorintian dal 2017.



# Programma del corso

- Oggetti e Prototype
- Array e metodi per l'iterazione
- Principali novità introdotte con ES6
- Javascript asincrono: Callback, Promise e Async / Await
- AJAX: connettersi a servizi remoti (XMLHttpRequest e Fetch API)
- Moduli: dividiamo il codice in più files
- Node, NPM e introduzione ai framework JS

# Veloce ripasso

## Primitive data types

Boolean

Null

Undefined

Number

BigInt

String

Symbol

## Reference data types

Objects

Arrays

Functions

Dates

MDN JavaScript data types and data structures

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data\\_structures](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures)

# typeof

L'operatore `typeof` ritorna una stringa che indica il tipo dell'operando.

```
console.log(typeof 42);
// expected output: "number"

console.log(typeof 'blubber');
// expected output: "string"

console.log(typeof true);
// expected output: "boolean"

console.log(typeof undeclaredVariable);
// expected output: "undefined"
```

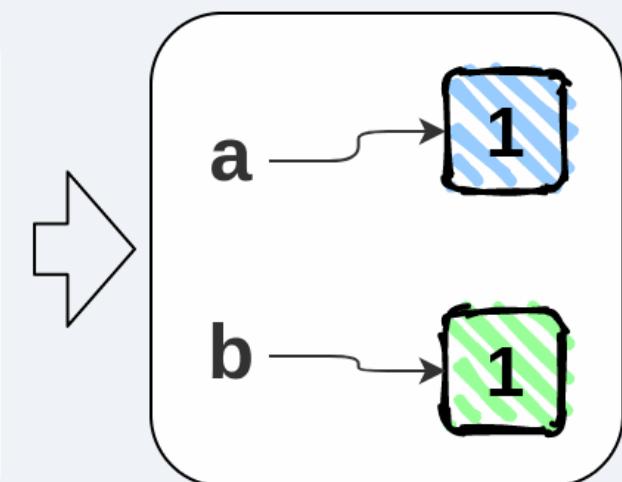
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof>

# Values VS. References

Ogni volta che assegnamo un valore ad una variabile, viene creata una copia di quel valore. Quando creiamo un oggetto, al contrario, stiamo creando una referenza a quell'oggetto. Se a due variabili viene assegnata la stessa referenza, i cambiamenti all'oggetto si rifletteranno su entrambe le variabili.

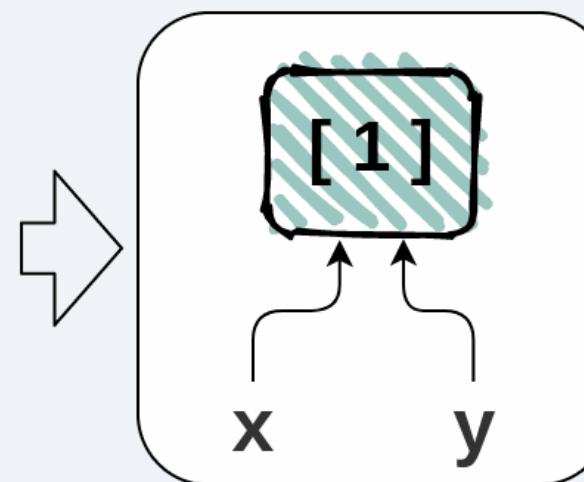
## Values

```
let a = 1;  
let b = a;
```



## References

```
let x = [1];  
let y = x;
```



# Oggetti

## Creare un oggetto

Object literal

```
const person = {
  name: ['Bob', 'Smith'],
  introduceSelf: function() {
    console.log(`Hi! I'm ${this.name[0]}.`);
  }
}
```

Constructor function

```
function Person(name) {
  this.name = name;
  this.introduceSelf = function() {
    console.log(`Hi! I'm ${this.name}.`);
  }
}
const frankie = new Person('Frankie');
```

# Oggetti

## Accedere alle proprietà

### Dot Notation

```
person.name  
person.introduceSelf()  
  
person.address?.city // Optional chaining
```

### Bracket Notation

```
person['name']  
person['introduceSelf']()  
  
const nome = 'name';  
person[nome];
```

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics>

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional\\_chaining](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining)

# Prototype

Il prototype è un oggetto associato di default in Javascript a tutte le funzioni e oggetti ed è il meccanismo tramite il quale gli oggetti Javascript ereditano le loro funzionalità.

È possibile aggiungere metodi e proprietà all'oggetto prototype. Questo permette agli altri oggetti creati dallo stesso costruttore di ereditare i metodi e le proprietà aggiunte.

[https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object\\_prototypes](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes)

# Oggetti e Es6

## Destructuring

Il destructuring è un'espressione JavaScript che consente di separare le proprietà di un oggetto (o i valori di un array) in variabili distinte.

```
const user = {  
  id: 42,  
  isVerified: true  
};  
  
const {id, isVerified} = user;  
  
console.log(id); // 42  
console.log(isVerified); // true
```

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

# Copiare un'oggetto

```
const originalObject = { original: 'original'}
```

## Shallow copy

- Spread operator or Object.assign()

```
const copiedObject = { ... originalObject};  
const copiedObject = Object.assign({}, originalObject);
```

## Deep copy

- JSON.parse(JSON.stringify())

```
const copiedObject = JSON.parse(JSON.stringify(originalObject));
```

- structuredClone (recentemente implementato, <https://developer.mozilla.org/en-US/docs/Web/API/structuredClone>)
- librerie (es. cloneDeep di Lodash <https://lodash.com/>)

# JSON (JavaScript Object Notation)

JSON è un formato di serializzazione di oggetti, array, numeri, stringhe, booleani e null. Utilizzato per lo scambio di dati, principalmente tra server e applicazione web, utilizza l'estensione .json

```
{  
  "name": "Mario",  
  "surname": "Rossi",  
  "active": true,  
  "favoriteNumber": 42,  
  "birthday": {  
    "day": 1,  
    "month": 1,  
    "year": 2000  
  },  
  "languages": [ "it", "en" ]  
}
```

# Funzioni e Es6

## Default parameters

I default parameters consentono di inizializzare i parametri della funzione con dei valori di default se non viene passato nessun valore oppure undefined.

```
function multiply(a, b = 1) {  
  return a * b;  
}
```

## Rest parameters

Questa sintassi consente alla funzione di accettare un numero indefinito di argomenti sotto forma di array.

```
function logItems( ... theArgs ) {  
  console.log(theArgs);  
}  
logItems(1, 2, 3); // expected output: [1, 2, 3]
```

# Arrow function

Una arrow function è un'alternativa compatta alle tradizionali funzioni. Si tratta di una funzione anonima che non ha un binding del this, e non dovrebbe essere utilizzata come metodo, oltre a non poter essere utilizzata come costruttore.

```
param => expression

(param1, paramN) => expression

(param1, paramN) => {
  let a = 1;
  return a + param1 + paramN;
}

const somma = (x, y) => x + y;
```

- Se è presente un solo parametro non necessitano delle parentesi tonde
- Se di una sola riga non necessitano di parentesi graffe e prevedono il return implicito

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow\\_functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

# Array

## Creare un array

```
const array = new Array(); // By using the new keyword  
  
const arrayLiteral = [value1, value2, ... .valueN]; // By using Array literals
```

## Accedere agli elementi

```
array[indice];
```

NB. L'indice del primo elemento è 0!

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

# Array e Es6

## Destructuring

Il destructuring è un'espressione JavaScript che consente di separare i valori di un array (o le proprietà di un oggetto) in variabili distinte.

```
const foo = ['one', 'two', 'three'];
const [red, yellow, green] = foo;
console.log(red); // "one"
console.log(yellow); // "two"
console.log(green); // "three"
```

## Spread operator

La spread syntax consente di "espandere" gli elementi di un array. Nella pratica è utilizzata per aggiungere elementi ad un array o oggetto, combinare array o oggetti e copiare array o oggetti (shallow copy).

... array

# Array.prototype.forEach()

Il metodo forEach esegue la funzione passata come parametro per ciascun elemento dell'array.

```
const array1 = ['a', 'b', 'c'];

for (i = 0; i < array1.length; i++) {
    console.log(array1[i]);
}

array1.forEach((element) => console.log(element));
```

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/forEach](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach)

# Array.prototype.map()

Il metodo map crea un nuovo array popolato con i valori ritornati dalla funzione passata come parametro. Questa funzione viene chiamata per ogni elemento dell'array originale.

```
const map = [1, 2, 3, 4, 5].map((number) => number * 2);

console.log(map);
// [2, 4, 6, 8, 10]
```

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map)

# Array.prototype.filter()

Il metodo filter crea un nuovo array contenente tutti gli elementi che passano il test implementato nella funzione passata come parametro.

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter(word => word.length > 6);

console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/filter](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter)

# Array.prototype.reduce()

Il metodo reduce esegue la funzione passata come argomento per ciascun elemento dell'array. Questa funzione, ad ogni iterazione avrà due parametri, il primo sarà il valore ritornato dal calcolo precedente mentre il secondo l'attuale elemento dell'array. Il risultato finale è un singolo valore.

```
const array1 = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4
const initialValue = 0;
const sumWithInitial = array1.reduce(
  (previousValue, currentValue) => previousValue + currentValue,
  initialValue
);

console.log(sumWithInitial);
// expected output: 10
```

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/reduce](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce)

# Classes

Le classi, introdotte con ES6, formalizzano il pattern di ereditarietà basato sulle classi che veniva simulato tramite funzioni e prototypes.

```
class Rectangle {  
  constructor(height, width) {  
    this.height = height;  
    this.width = width;  
  }  
}  
  
const rect = new Rectangle(20, 30);
```

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>

# Maps

L'oggetto Map contiene delle coppie di chiave-valore. Qualsiasi valore (sia oggetti che primitivi) possono essere utilizzati sia come chiavi che valori.

```
const map1 = new Map();

map1.set('a', 1);
map1.set('b', 2);
map1.set('c', 3);

console.log(map1.get('a')); // expected output: 1

map1.set('a', 97);

console.log(map1.get('a')); // expected output: 97

console.log(map1.size); // expected output: 3
```

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map)

# Sets

Gli oggetti Set sono collezioni di valori. Ogni valore può essere presente una sola volta, ciò rende il Set una collezione di valori univoci.

```
const mySet1 = new Set()

mySet1.add(1)          // Set [ 1 ]
mySet1.add(5)          // Set [ 1, 5 ]
mySet1.add(5)          // Set [ 1, 5 ]
mySet1.add('some text') // Set [ 1, 5, 'some text' ]

mySet1.has(1)          // true
mySet1.has(3)          // false, since 3 has not been added to the set

mySet1.size            // 5
```

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set)

# Bonus track: Ternary operator

L'operatore ternario è l'unico operatore Javascript che accetta tre operandi: una condizione seguita dal punto esclamativo (?), un'espressione da eseguire se la condizione è truthy seguita da due punti (:), e infine l'espressione da eseguire se la condizione è falsy. Questo operatore è frequentemente utilizzato come alternativa a if.

```
const risultato = condizione ? espressioneTrue : espressioneFalse;
```

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional\\_Operator?retiredLocale=it](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator?retiredLocale=it)

# Synchronous vs Asynchronous

La programmazione sincrona esegue un task alla volta e solamente quando il task precedente è completato viene eseguito il task successivo.

## Execution Context

L' Execution Context è un concetto astratto dell' ambiente in cui il codice JavaScript code viene interpretato ed eseguito. Ogni volta che del codice JavaScript viene eseguito, ciò avviene all'interno dell'execution context. Il codice di una funzione viene eseguito all'interno del function execution context, mentre il codice globale dentro il global execution context. Ciascuna funzione ha il suo execution context.

## Call stack

Il call stack è uno stack con una struttura LIFO (Last in, First out), utilizzato per memorizzare tutti gli execution context creati durante l'esecuzione del codice.

# Synchronous vs Asynchronous

La programmazione asincrona è una tecnica che consente al programma di iniziare un potenziale task di lunga durata e, invece che attendere l'esecuzione del task, continuare ad eseguire altri task mentre il precedente è in esecuzione.

<https://www.freecodecamp.org/news/synchronous-vs-asynchronous-in-javascript/>

<https://blog.bitsrc.io/understanding-asynchronous-javascript-the-event-loop-74cd408419ff>

## **setTimeout()**

setTimeout() è una funzione asincrona che esegue un blocco di codice o una callback con un ritardo impostato in millisecondi.

# Event handlers e callbacks

Gli Event handlers sono a tutti gli effetti una forma di programmazione asincrona. Non è possibile sapere in anticipo quando l'utente cliccherà un certo elemento, quindi definiamo un event handler per l'evento click. Questo event handler accetta una funzione che verrà chiamata al manifestarsi dell'evento.

## Callbacks

Una callback non è altro che una funzione che viene passata ad un'altra funzione, che la eseguirà al momento opportuno. Le callbacks sono state il principale modo in cui la programmazione asincrona è stata implementata in JavaScript.

<https://nodejs.dev/learn/javascript-asynchronous-programming-and-callbacks>

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing>

# Promises

Le Promises sono la base della programmazione asincrona del moderno JavaScript.

Una promise è un oggetto ritornato da una funzione asincrona che rappresenta lo stato attuale dell'operazione. Nel momento in cui la promise viene tornata, l'operazione non è ancora stata ultimata ma fornisce dei metodi che consentono di gestire l'eventuale successo o fallimento dell'operazione.

Una promise può essere in uno dei tre seguenti stati: pending, resolved, o rejected.

```
const promise = new Promise((resolve, reject) => {
  const res = true; // An asynchronous operation.
  if (res) {
    resolve('Resolved!');
  } else {
    reject(Error('Error'));
  }
});
```

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Promises>

# Promises

## .then()

Il metodo .then() viene utilizzato per ricevere l'eventuale risultato (o errore) dell'operazione asincrona. then() accetta due funzioni come argomenti. La prima viene chiamata nel caso in cui la promise viene risolta, la seconda se la promise viene rigettata.

```
promise.then((res) => {
  console.log(value);
});
```

## .catch()

Se viene aggiunto catch() alla catena della promise, verrà chiamato nel caso in cui l'operazione asincrona fallisce. Risulta utile nel caso in cui si intenda concatenare più operazioni asincrone.

```
promise.catch((err) => {
  alert(err);
});
```

# Promises

## .finally()

Se viene aggiunto il metodo finally() alla catena della promise, verrà chiamato in seguito al completamento della promise sia nel caso in cui venga risolta che nel caso in cui fallisca.

## .all

Alle volte è necessario combinare delle promise, invece che eseguirle una dopo l'altra. Per questo esistono degli helpers. Ad esempio, a volte, è necessario che più promise vengano risolte ma non dipendono una dall'altra. In un caso del genere è possibile lanciarle tutte contemporaneamente ed essere poi notificati quando tutte le promises vengono risolte. Promise.all() consente proprio questo, accetta un array di promises, e ritorna una singola promise con un array di valori nel metodo then.

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)

# Async / Await

La sintassi async...await offre un nuovo modo più leggibile e scalabile di gestire le promises.

Una funzione asincrona può essere creata con la keyword `async` prima del nome della funzione o delle parentesi `()` nel caso di arrow function. L'operando `await` è una promise. Quando il codice incontra la keyword `await`, l'esecuzione della funzione `async` è messa in pausa ed attende la risoluzione della promise. `Await` può essere utilizzato solamente in una funzione con `async`.

```
function resolveAfter2Seconds() {
  return new Promise(resolve => setTimeout(() => resolve('resolved'), 2000));
}

async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  console.log(result); // expected output: "resolved"
}

asyncCall();
```

Le funzioni `async` utilizzano `try...catch` per gestire gli errori.

# AJAX

Asynchronous JavaScript and XML, è un acronimo coniato nel 2005 da Jesse James Garrett, che descrive un "nuovo" approccio all'utilizzo di tecnologie esistenti insieme che includono HTML or XHTML, CSS, JavaScript, DOM, XML, XSLT, ed il più importante, l'oggetto XMLHttpRequest object. Tramite queste tecnologie le web applications sono in grado di eseguire rapidi ed incrementali aggiornamenti dell'interfaccia utente, senza ricaricare l'intera pagina.

XMLHttpRequest può inviare e ricevere informazioni in diversi formati (tra cui JSON, XML, HTML, and text files).

Le due principali funzionalità di AJAX sono:

- Effettuare richieste ad un server senza ricaricare la pagina
- Ricevere e gestire dati da un server

<https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX>

# XMLHttpRequest

Per inviare una richiesta HTTP ad un server utilizzando JavaScript, è necessario creare un oggetto XMLHttpRequest, "aprire" un url ed inviare la richiesta. XMLHttpRequest mette a disposizione una serie di eventi per "seguire" il processo della richiesta.

```
const xhr = new XMLHttpRequest();

xhr.onload = function(){
    // Process the server response here.
};

xhr.open("GET", "mysite.com/api/getjson");
xhr.send();
```

[https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Using\\_XMLHttpRequest](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest)

# Fetch API

L'API Fetch è utilizzata per effettuare richieste HTTP requests utilizzando le Promises. La funzione principale `fetch()` accetta un parametro URL e ritorna una promise che può essere risolta con i risultati della nostra chiamata o rigettata (ad esempio in caso di errori di rete).

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

La funziona `fetch()` accetta un secondo argomento opzionale, un oggetto contenente delle opzioni per customizzare la request. Può essere utilizzato per modificare la request type, gli headers, specificare un request body, e molto altro.

[https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)

<https://jsonplaceholder.typicode.com/>

<https://github.com/public-apis/public-apis>

# Moduli

Con l'evolversi di Javascript, le applicazioni sono diventate sempre più complesse richiedendo sempre più codice, per questo si è reso necessario pensare a dei modi per dividere il codice in più moduli che possono essere importati all'occorrenza. Node.js consente di utilizzare i moduli e tante altre librerie hanno permesso nel corso degli anni agli sviluppatori di sfruttare questa caratteristica (ad esempio CommonJS e le librerie basate su AMD come RequireJS, e più recentemente Webpack and Babel).

I browser moderni hanno iniziato a supportare questa funzionalità in maniera nativa.

```
// index.html
<script type="module" src="main.js"></script>

// main.js
import { name } from './modules/name.js';

// modules/name.js
export const name = 'square';
```

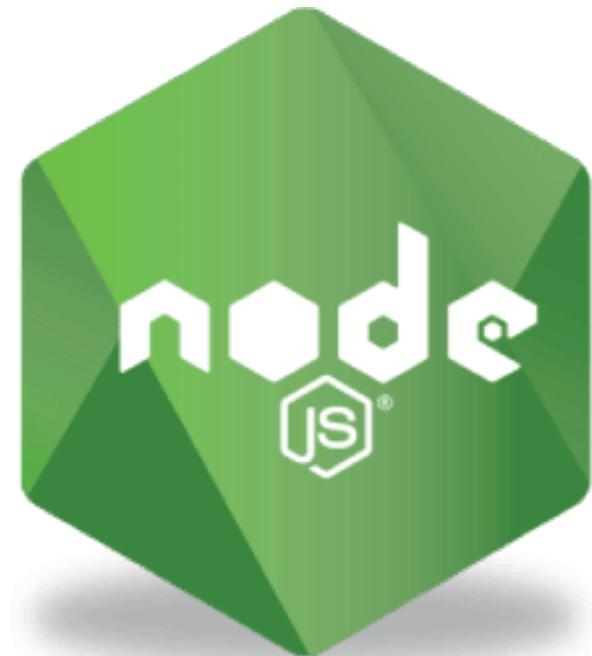
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules>

# Node.js

Node.js è un runtime Javascript open-source costruito sul motore V8 di Chrome. Consente di eseguire codice Javascript fuori dal browser.

Può essere utilizzato per scrivere applicazioni server-side con accesso al sistema operativo ed al file system.

Quando le nuove funzionalità di Javascript vengono supportate da V8, possono essere utilizzate anche con Node.



<https://nodejs.org/>

# NPM

NPM – o "Node Package Manager" – è il package manager di default di Node.js.

NPM consiste principalmente in:

- una CLI (command-line interface) utilizzata per pubblicare e scaricare librerie;
- un repository online di librerie Javascript



Per inizializzare un progetto tramite npm utilizzare il comando ``npm init`` che creerà un file ``package.json``, contenente una descrizione del progetto.

Per maggiori informazioni: <https://docs.npmjs.com/cli/v8/configuring-npm/package-json>

Il comando principale per installare una libreria tramite npm è: ``npm install <nome libreria>``

<https://www.npmjs.com/>

# Frameworks JavaScript

I frameworks JavaScript sono una parte essenziale del moderno sviluppo web che forniscono agli sviluppatori strumenti per creare applicazioni web interattive e scalabili.

I principali framework front-end:

- React
- Angular
- Vue
- Svelte

Back-end:

- Express

[https://developer.mozilla.org/en-US/docs/Learn/Tools\\_and\\_testing/Client-side\\_JavaScript\\_frameworks](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks)

<https://roadmap.sh/frontend>

# Vue

*Vue (pronounced /vju:/, like view) is a JavaScript framework for building user interfaces. It builds on top of standard HTML, CSS and JavaScript, and provides a declarative and component-based programming model that helps you efficiently develop user interfaces, be it simple or complex.*

Versione attuale: 3. Per iniziare un progetto con Vue:

```
npm init vue@latest
```

e seguire le istruzioni della CLI.



<https://vuejs.org/>

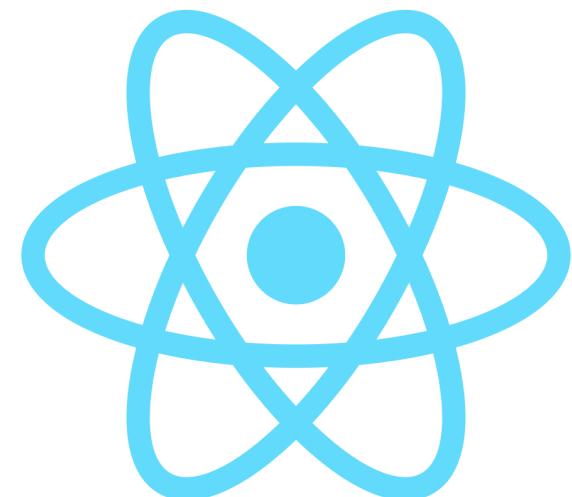
# React

*Una libreria JavaScript per creare interfacce utente*

Versione attuale: 17.0.2. Per iniziare un progetto con React utilizzare la CLI ufficiale `create-react-app`  
[\(https://create-react-app.dev/\)](https://create-react-app.dev/):

```
npm init react-app my-app
```

e seguire le istruzioni della CLI.



<https://it.reactjs.org/>

# Angular

*The modern web developer's platform*

Versione attuale: 13.3.0. Per iniziare un progetto con React utilizzare la CLI ufficiale @angular/cli (<https://angular.io/cli>):

```
npm install -g @angular/cli  
ng new my-first-project
```

e seguire le istruzioni della CLI. La CLI di Angular consente anche di generare componenti, moduli, servizi, direttive, ecc...



<https://angular.io/>

# Deployare la vostra applicazione

- Netlify (<https://www.netlify.com/>)
- GitHub Pages (<https://pages.github.com/>)
- Firebase Hosting (<https://firebase.google.com/>)
- Heroku (<https://www.heroku.com/>)
- ...